MILLERS GRAPHICS

FEBRUARY 1984
$1.50

THE SMART PROGRAMMER

To start off the first issue of The Smart Programmer monthly newsletter we would like to THANK everyone from around the world who subscribed and helped make it all possible. Since February 84 is the first issue the last issue for the first year will be January 85.

We would also like to say THANK YOU for all the complimentary letters and questions. We read everyone of them and we do appreciate your comments. Please don't stop sending them for they truly help us decide on topics for the newsletter. Well that is enough editorial for now so lets start the fun stuff with the first monthly column.

Q&A

We have received a number of questions on proprietary disk protection.

Proprietary Protection is used on Scott Adams, Plato and other third party diskettes to prevent you from making a backup copy. It is placed on the diskette when it is initialized and prevents the disk from being copied with the BACKUP DISK feature of the Disk Manager command module.

To enter the Proprietary Protect mode of the Disk Manager command module go to any of the menu selection screens in the module. Then hold down the FCTN key and press X ten times. On the tenth time you will hear a beep and > < will appear at the top of the screen. Now when you initialize your diskette Proprietary protection will be placed on it.

I do not recommend that you use this for your everyday disks since they can no longer be backed up with the disk manager command module. If you re-initialize the diskette in the normal fashion Proprietary protection will be removed, but so will your programs and files so use this with caution and always maintain a backup diskette that is NOT Proprietary protected!!

It would not be fair to all of the software authors who are supporting the 99/4A to tell you how to remove this protection without losing the data on the disk so we will have to leave that up to you to figure out. I will, however, say that it can not be removed with the disk manager module. One last thing before we leave this topic. You can get out of the Proprietary mode by either going back to the title screen and starting up the disk manager over again or you can press FCTN X one more time and the > < will be erased from the top of the screen.

We have also received a number of questions regarding software.

Many of the questions asked what we thought about a particular piece of software written by some other company. I don't feel that it is fair for one software company to review another company's work, but there will be times when we will discuss a utility program and its' practical applications.

Some of the questions were regarding our policy on publishing and producing software for sale through Millers Graphics. On this subject let me say that we are not currently looking for any new titles. We are however looking for short programs that exploit the power of the 99/4A to be

published in future issues of the newsletter especially if they use PEEK's and/or LOAD's.

There were also a few inquiries as to whether or not we would be publishing any large programs, 4K+. One that comes to mind is from a gentleman who has a word processor that works with the minimum 16K system and a cassette recorder. I told him that I would ask all of you to see if you would like to have this type of programming reproduced in the pages of the Smart Programmer, how about it? Write and let us know, just drop us a postcard with "LONG IS O.K." or "KEEP THEM SHORT." on it and we will take it from there.

If you leave it up to me, since I love the numbers, we will all be neck deep in memory maps, operating systems, GPL language, DSR's, Disk formats, CALL PEEK's and CALL LOAD's before long, so please write.

Many of the questions were on SPRITE Coincidence and how to detect it with various other sprites, locations and/or graphic characters and if there is a CALL PEEK that can be used for faster detection.

VDP RAM is separate from the CPU, central processing unit, memory so we can not directly PEEK or LOAD into VDP RAM from Extended Basic. In Extended Basic we are quite limited on the amount of sprite information we can obtain from CPU RAM. The few addresses that are in the Smart Programming Guide for Sprites that relate to sprites are located in CPU RAM and this is the only information we can directly obtain at this time.

When we start mapping out VDP RAM we will publish PEEKV and POKEV routines that you can use through CALL LINK if you have Expansion Memory. If you have the Mini-Mem or Editor Assembler module you can PEEKV and POKEV from Basic although Basic does not recognize auto motion sprites. Before we get in too deep here, I would like to save the answers to these questions for the issue that contains the VDP memory maps. I think that it can be better explained when you have the whole picture in front of you.

Many people have asked for help on Adventure games.

The best adventure game I can play is the one that deals with all the numbers stored in memory and how they relate to one another and I am afraid that this one takes up most of my game time. Even when I try to play a game I find myself trying to figure out how the program works and consequently I usually lose very rapidly. So, based on this I am afraid that I can't be much help on solving a particular adventure.

On the other hand I can give you a little trick that might help. The Adventure command module contains the input prompts and the list of acceptable responses. It also contains some of the decision making based on your response. With this in mind it stands to reason that each of the cassette and diskette games that use this module have a similar format. Also the games that you have partially played and saved contain the same format from scenario to scenario.

The point I am getting to is that you can load one game and load your progress from another scenario and run it. Naturally the game will no longer make any sense but it will from time to time return a few hints for you to try when you load the game and your progress the regular way. Don't forget to check your inventory before you reload the game the proper way, you may find some items listed that you didn't even know existed. Happy Adventuring, I'll go back to the numbers, I find them to be a lot easier.

A gentleman from Georgia recently asked us how to get a Basic program that is stored on disk and is too large to run with the disk drives attached to run. He was referencing the CALL LOAD in the first volume of the Smart Programmer that shuts off the disk drives. At the time that I talked with him I said I didn't know but after I thought about it a little while some tips that Paul Schippnick gave us came to mind.

The problem was such that the program could be loaded into memory from the disk but it contains a large number of numeric and string variables which eat up a lot of

memory when the program is RUNning. In Basic the computer does not recognize Expansion Memory as a place to load programs so they must load and run from VDP RAM. Unfortunately the disk buffer space is also in VDP RAM and uses up some of the space that may be necessary to run large Basic programs. Even with the CALL FILES(1) command there is still aprox 500 bytes retained as a disk buffer and this may be just enough to crash the program. The CALL LOAD, which is listed below, will shut off the disk drives but in Basic you must type in NEW to open up the memory space. Naturally whenever you type in NEW your program is cleared out of memory and that was the problem.

The solution requires the Mini-Memory module to be plugged in the cartridge port and Expansion Memory to be attached and turned on. With the Mini-Mem in the port there are a few new commands added to the Basic language, even though you have selected TI BASIC. The ones that we are concerned with are SAVE and OLD, MINIMEM, EXPMEM1 and EXPMEM2 and out of these we really only need SAVE and OLD EXPMEM2. The procedure for running these large Basic programs from disk is as follows:

1. Power Up, select Basic and type in CALL INIT. This will initialize memory expansion and it will also clear out what ever was stored in the Mini-Mem module. If you want to retain what is in the Mini-Mem just turn off the memory expansion and then turn it back on and that will clear it out without erasing the Mini-Mem.

2. Load the program you want to run from your disk. OLD DSK1.xxxxxxxxxx .

3. Type in SAVE EXPMEM2 and press ENTER. This will copy the program in VDP RAM into the exp-memory.

4. Type in CALL LOAD(-31888,63,255) and press ENTER. This tells the computer not to reserve any room in VDP RAM for the disk buffers.

5. Now type in NEW and press enter. You have just opened up the extra memory space in VDP RAM that the disk was reserving. But since Basic does not recognize expansion memory your program is still intact in high exp-memory, addresses hex A000 through FFFF.

6. Just type in OLD EXPMEM2 and press enter and this will copy the program in the exp-memory back into the expanded VDP RAM program area. A copy of the program is still in the exp-memory and it will remain there until you turn off the exp-memory, save another program into that space or type in CALL INIT. No, you can't MERGE Basic programs.

7. Type in RUN and press ENTER and the program should now run without giving you a MEMORY FULL ERROR.

We didn't use SAVE MINIMEM or SAVE EXPMEM1 because neither one of these spaces are large enough to store a large Basic program, 12K+. The MINIMEM space is 4K of RAM, The EXPMEM1 space is 8K of RAM and the EXPMEM2 space is 24K of RAM. Also the EXPMEM1 space will allow you to save a program out there but it doesn't like OLD EXPMEM1 so you can't bring it back. You can use any one of these three names in an OPEN statement for files provided you are not LINKing to an Assembly language subroutine in which case it is best to just use EXPMEM2 in your OPEN statements for files.

The best solution, if you have exp-memory, is to rework the programs to allow them to be loaded directly into exp-memory via Extended Basic. You will probably have to rework some of the CALL CHAR's, HCHAR's, VCHAR's and CALL COLOR's to compensate for the lack of character sets 15 and 16 in Extended Basic. You will however find that they run a little faster and they are easier to edit in Extended Basic.

GENERAL TIPS & TRICKS

Mike McCue in New York, who we are working with on our new book, gave us the following trick.

You can change the text and or screen color while you are in the immediate mode or program entry mode by using the following sequence. Note: these commands are typed in without using line numbers and you can replace any of the color numbers with any of the colors that you would like.

To change just the screen color type in;
CALL SCREEN(6)::ACCEPT AT(1,1):A

After you have typed this in press enter, the screen will change to blue and the cursor will be at row 1, column 1, waiting for you to enter a number. DO NOT press enter, DO NOT enter a number just press FCTN 4 (CLEAR). The cursor will jump back down to the bottom of the screen and you will be back in the immediate mode and you can now type in or list your programs against the new background, screen, color.

The only problem with this is that if you create an error, execute CALL CLEAR, PRINT or RUN your program the screen will return to its normal cyan color.

To change the cursor, text and screen color type in
FOR I=0 to 12::CALL COLOR(I,16,1)::NEXT I::
CALL SCREEN(5)::ACCEPT AT(1,1):A
Then follow the above instructions.

To change just the cursor color type in
CALL COLOR(0,7,1)::ACCEPT AT(1,1):A
Then follow the above instructions.

There is one other trick that you can use with ACCEPT AT but it is not very reliable. Sometimes you can get an ACCEPT AT statement to accept character input out to the 32nd column by using a subscripted variable with a simple formula as the subscript for the input variable. For example:

```
100 ACCEPT AT(1,1):A(2-1)
```

It is not dependable enough to use on a regular basis but it does show you that there are a few bugs in every language. If you avoid using formulas in your ACCEPT AT subscripted variables you won't have the problem of erratic line lengths.

One last note on these and many of the other tips, tricks and items that we will be publishing in future issues. There are approximately 5 to 7 different operating systems in the various 99/4 and 99/4A consoles. There are also a few slightly different versions of Extended Basic out there. We try to test as much of this as possible but you may find from time to time that somethings may not work exactly the same on your system.

OOPS!

Looks like we goofed on a couple of items in our VOL. 2 newsletter.

In our column on CorComp Inc. we miss quoted the prices on the cards for the Expansion Box. The 32K memory expansion card has a suggested retail price of 149.95 and the RS232 Interface card carries a suggested retail price of 99.95.

Also we were told that the RS232 card would be FULLY compatible with the TI card and that the parallel port would be modified slightly to make it a true Centronics compatible output. You see the TI RS232 card is not a true parallel Centronics output and a few of the printers out there need a special cable with some converters built into the connector to allow it to work. CorComp was going to change this around when they built their card but it appears that it wasn't done on the first production run. They also changed pins 2 & 3 around on the serial output for some unknown reason. This is not a major problem you just have to swap wires 2 & 3 on one end of your cable, but it would have been nice if it was fully compatible. Maybe they will change this on future cards? I hope so.

We also have a bad value in the number conversion program on page 3. Line number 130 should read:

```
130 INPUT "DEC #=":DEC :: IF
 DEC<-32768 OR DEC>65535 THE
N 130 ELSE A,DEC=INT(DEC-655
36*(DEC<0)):: GOSUB 200 :: G
OSUB 220 :: GOTO 160
```

The OR DEC>65536 test caused the program to crash if you input 65536 as your decimal number to be converted.

Our last goof is concerning the CALL LOAD address of -31806. We stated that if you turn off the auto sound processing by loading this address with 32 that you will lock up your computer if you execute another CALL SOUND. Well that was only partially right. If you use sound statements with positive durations such as CALL SOUND(100,660,0) it will lock up but you can use negative durations such as CALL SOUND(-100,660,0) without locking up.

PEEKING AROUND

This will be another one of our monthly columns in which we will be publishing memory maps, peek and load addresses and other items relating to memory locations.

So lets get start by looking at a couple of new addresses that were sent to us from Mike Sijacic of Michigan. The first address is hex >8373 or decimal -31885. This is located in the scratch pad RAM area and it is an address that is known as the " Least significant byte of the subroutine stack pointer". Mike stated that if you load 255 into this address that it acts the same as typing in BYE. Unfortunately it does not work on all the systems out there but here is another way to accomplish the same effect.

```
100 CALL INIT :: CALL PEEK(2
,A,B):: CALL LOAD(-31804,A,B
)
```

The word at memory location 0 is the workspace pointer for FCTN + (QUIT), the power up routine. The word at memory location 2 is the address for the power up routine. So what we have done is to get the address for the power up routine and load it into the Interrupt Service Routine (ISR) hook. Then on the next VDP Interrupt, which happens 60 times a second, the computer will jump to the address in the ISR hook location to continue execution and since we have loaded the address for the power up routine the computer acts as if you pressed FCTN +.

The next address that Mike sent us works fine and what it does is to make the computer restart Extended Basic. It acts the same as when you select Extended Basic from the menu so it will try to boot a file named LOAD if you have a disk drive attached to your computer. This address is also located in scratch pad RAM at hex >8326 or decimal -31962. This address is known to Extended Basic as "the return address from Assembly Language code". Loading just about any value here will restart Extended Basic.

```
CALL LOAD(-31962,255)
```

The last address that Mike sent is at >83FC or decimal -31748. This address is bit mapped and it changes the speed of your sound statements and the rate of flashing on the cursor. CALL LOAD(-31748,1) is normal and anything else effects the cursor and sound speed. Use it with caution and we will talk more about it in a future issue.

To start off the memory maps we have enclosed an Overall System map. The Overall System map contains all of the addresses that are directly accessible by the TMS 9900 microprocessor. Along with this 64K block of memory our systems contain an additional 16K block of VDP RAM that is accessed through the TMS 9918A video display processor. Our system is also currently set up to access up to 48K of GROM, Graphics ROM, but with a little additional hardware it can access up to 498K of GROM. We will talk more about this in future issues when we discuss GROM and GPL, Graphics programming language. Right now lets break down the Overall System map.

ROM - Read Only Memory. You can only read, CALL PEEK, from this memory, you cannot write, CALL LOAD, to it.

ROM comes in many forms, some of which are ROM, PROM, EPROM, EEPROM and GROM, but they all serve a similar basic function and that is to store information. This information is in the form of numbers and in our systems the numbers range from 0 to 255 in each address. These numbers are used by the CPU, TMS 9900 microprocessor, as programs, subprograms, data and vector or jump tables.

One of the main differences between the different types of ROM is how they are programmed. ROM and GROM are generally programmed with a mask at the time of manufacturing by etching the program right into them. PROM stands for Programmable ROM, and they are programmed through a PROM, EPROM or EEPROM programmer. This process is commonly called "Burning a PROM". Once a PROM is burned it cannot be erased, on the other hand EPROM's, Eraseable Programmable ROM's, can be erased by exposing the chip to strong ultraviolet light, a sun lamp, for about 15 minutes. EEPROM's, Electronically Erasable and Programmable ROM's, can be erased with a special electroinc signal.

Our systems mainly contain ROM's and I haven't found ANY EEPROM's so you do not have to worry about wiping out your system's programming by typing in the wrong thing. As a matter of fact, over the past five years I have yet to find any command or commands that have hurt my systems at all. The worst I've done is to lock them up but that is easily recovered from by turning the console off and then back on again. So feel free to type in whatever you want, you can't hurt it.

One last note on ROM, it is a nonvolatile type of storage which means that it retains what ever was programmed into it even when the power is shut off. Without any ROM and GROM in our computers we could only communicate with them in machine language, until we loaded some smarts into their RAM.

RAM - Random Access Memory. You can read, CALL PEEK, and write CALL LOAD to this type of memory chip. RAM is a volatile type of memory in that it forgets whatever was programmed into it when the power is shut off.

This is where most of the activity goes on when you are programming or running a program. We have 4 areas of RAM in our computers, VDP RAM, Low memory expansion, the area for memory mapped devices and the scratch pad and finally High memory expansion. It is also possible to have RAM in the Cartridge Port and DSR areas of memory. Lets look at the different areas of memory and how they are used in our systems.

>0000 - >1FFF ROM 8K Bytes

Hex 0 through 1FFF is the area of CPU memory that is reserved for ROM in our systems and contains the following:

The Interrupt Vectors and interrupt processing for Auto sound, Sprite motion, Interval timer and DSR interrupts.

The XOP (Extended Operation instruction) vectors.

The keyboard scanning routine.

The GPL (Graphics Programming Language) interpreter.

The low level cassette DSR (Device Service Routine).

Part of the Basic interpreter.

The Radix 100 floating point routines for addition, subtraction, multiplication and division.

The subprogram and DSR search routine.

It is this section of memory along with GROM chip 0 that takes control when we turn the console power on.

>2000 - >3FFF RAM 8K Bytes

Hex 2000 through 3FFF is mapped out for Low memory expansion. When you plug the 32K memory expansion card in 8K of it is mapped into this space and the other 24K is mapped at a higher address. If you do not execute CALL INIT this space will not be used by Extended Basic. When CALL INIT is executed the Extended Basic command module loads some Assembly language routines into this area. This space is used by the Editor Assembler command module and when you select this module from the menu the Assembly language routines are automatically loaded into this space.

>4000 - >5FFF ROM 8K Bytes

Hex 4000 through 5FFF is the area of our memory that is mapped out for the DSR's, Device Service Routines. This area is zeroed out unless you are accessing the RS232 card, disk controller, P-Code card or the Video controller peripheral.

A DSR is nothing more than the programming necessary to communicate with one of these devices. When you access one of these cards, such as OLD DSK1.xxxx, the DSR for that card is paged into this space for the CPU to access it. This paging is handled by the TMS 9901 Programmable Systems Interface chip which takes care of the interrupt and Input/Output interface functions. The DSR is only paged into this area during the exact moment that you are addressing the card, such as OPEN #1:"RS232" or PRINT #1:" HELLO" and then it

is paged back out so you can not peek into this area from Extended Basic and see the DSR. In a future issue we will show you how to use the Debugger from the Editor/Assembler to look at the DSR's and everything else for that matter.

>6000 - >7FFF ROM and/or RAM 8K Bytes

Hex 6000 through 7FFF is the area of memory that is mapped out for the Command Module's ROM and/or RAM. Some of the command modules such as the Disk Manager and Editor/Assembler do not contain any ROM or RAM so with these modules this area is zeroed out.

The Extended Basic module contains 12K of ROM and it is mapped into this area. How, you might be asking, do they fit 12K into an 8K space? They do it by flip/flopping (paging) the 4K of ROM at >7000 - >7FFF. This is accomplished by writing to ROM address >65A6 and it is flipped back by writing to ROM address >7000. This is one of the few times that executing a CALL LOAD to a ROM address has any effect. If you execute CALL INIT :: CALL LOAD(26022,1), you will find that your computer locks up and you will have to shut it off and back on to regain control and this is because you flipped part of the Extended Basic language at the wrong time.

If you have the Mini-Memory module plugged in then >6000 - >6FFF (4K) is ROM and >7000 - >7FFF (4K) is Mini-Mem RAM. As you can see this block of memory varies with each of the modules that are plugged in. One last note on this space, even though the cartridge is mapped into this space the GROM in the cartridges is accessed from a different location in memory.

>8000 - >9FFF RAM 8K Bytes

Hex 8000 through 9FFF is the area of memory used for memory mapped devices such as VDP RAM, GROM, the Sound & Speech processors and it is also used for the CPU's scratch pad area.

Scratch Pad RAM is the area of memory that the CPU uses to hold a wide variety of temporary and system information. The scratch pad occupies 256 bytes of a 1K block. This 1K block is not fully decoded so the scratch pad repeats itself every 256 bytes within this block. In our systems the scratch pad is at >8300 through >83FF but you can find the same information at >8000 - >80FF, >8100 - >81FF and >8200 - >82FF.

This area of RAM operates very fast because unlike the Expansion RAM which is on an 8 bit bus this area of RAM is on a 16 bit bus. If you write Assembly language programs you should try to keep your workspace within this 256 byte area but be careful of the precautions called out in the Editor/Assembler manual on pages 404 thru 406. Because this area of memory has so much going on in it we will be mapping it out in a future issue.

The remainder of the items between >8400 and >9FFF are known as memory mapped devices. These items are accessed through a small window, address, and they transfer one byte at a time, to or from, the device until all of the requested bytes have been transferred. Some of the devices have separate addresses for reading/writing data and setting up the address within the device to be read from or written to such as VDP RAM and GROM. While others may use special coding right in with the data stream to determine whether it is a read or write operation and what the address is within the device being accessed.

It is because VDP RAM is a memory mapped device and because TI did not provide PEEKV and POKEV statements in Extended Basic that we can not peek and poke or load into it.

>A000 - >FFFF RAM 24K Bytes

Hex A000 through FFFF is the area of memory used for High Memory Expansion. If you have memory expansion attached to your computer this is where your Extended Basic programs are loaded into and RUN from. This area also holds the line number table and numeric values for the numeric variables. The line number table is more or less a set of pointers that tell the CPU what address in memory a specific Extended Basic program line starts at. It is because the Low and High memory expansions are split up in the memory map that we can not load an Extended Basic program larger than 24K into our computers.

OVERALL SYSTEM MAP

>0000 >1FFF	**CONSOLE ROM** Interrupt Vectors, XOP Vectors, GPL Interpreter, Floating Point Routines, XMLLNK Vectors, Low-level cassette DSR etc.	8K Bytes
>2000 >3FFF	**LOW MEMORY EXPANSION RAM** Varies according to the loader used (Assembly). Generally not used by Extended Basic programs.	8K Bytes
>4000 >5FFF	**DSR ROM** Device service routines Determined by CRU bit setting Disk Controller, RS232, P-Code etc.	8K Bytes
>6000 >7FFF	**CARTRIDGE PORT** ROM (& Mini-Mem RAM) 12K of Extended BASIC ROM. Upper 4K @ >7000 - >7FFF is flipped to page in another 4K for a total of 12K	8K Bytes
>8000	**RAM MEMORY MAPPED DEVICES** VDP, GROM, SOUND & SPEECH	8K Bytes
>8000 >80FF	Duplication of scratch pad ram @ >8300 ->83FF	
>8100 >81FF	Duplication of scratch pad ram @ >8300 ->83FF	
>8200 >82FF	Duplication of scratch pad ram @ >8300 ->83FF	
>8300 >83FF	CPU SCRATCH PAD RAM 256 bytes	
>8400 >87FF	SOUND CHIP	
>8800	VDP READ DATA	
>8802	VDP STATUS (MSBy)	
>8BFF		
>8C00	VDP WRITE DATA	
>8C02	VDP READ/WRITE ADDRESS (to write set MSb of the MSBy to 01)	
>8FFF		
>9000 >93FF	SPEECH READ	
>9400 >97FF	SPEECH WRITE	
>9800	GROM/GRAM READ DATA	
>9802	GROM/GRAM READ ADDRESS	
>9BFF		
>9C00	GROM/GRAM WRITE DATA	
>9C02	GROM/GRAM WRITE ADDRESS	
>9FFF		
>A000 >FFFF	**HIGH MEMORY EXPANSION RAM** Extended Basic High Memory Usage, Free space end pointed to by CPU RAM PAD address >8386 Numeric values --- Line number table --- X-Basic program space	24K Bytes

ADDITIONAL MEMORY NOT IN THE CPU ADDRESS SPACE

VDP RAM >0000 - >3FFF 16K Bytes

This is the memory that comes built into the console and it is separate from the rest of CPU memory. If you do not have memory expansion attached to your computer this is where your Extended Basic programs reside and run from. Basic does not recognize memory expansion so your Basic programs always reside here, also a pure Assembly language program cannot be run from this space.

The TMS 9900 CPU addresses this space through the TMS 9918A VDP processor. This space is mapped out according to the language you are running in and the VDP mode you are in. In Extended Basic without memory expansion this area contains the following:

>0000 ->02FF	Screen Image Table
>0300 ->036F	Sprite Attribute Table
>0370 ->077F	Sound buffer, Crunch buffer, Roll out area for Floating point routines, Pattern descriptor table and Sprite descriptor table.
>0780 ->07FF	Sprite motion table.
>0800 ->081F	Color table.
>0820 ->35D7	PAB's, Numeric Values, Line Number table, Program space, and String space.
>35D8 ->3FFF	Disk buffer space.

If you have memory expansion then >0820 ->35D7 is used for the PAB's and the String space. The Numeric values, Line Number table and Program space are moved into memory expansion at >A000 ->FFFF after the "PROGRAM" type file has been loaded from cassette or disk.

This area of memory is quite extensive so we will be devoting a couple of issues to mapping it and the VDP registers out for Basic, Extended Basic and the Editor Assembler. We will also map out TEXT and Bit-Mapped modes so gather up your VDP questions and send them in so we can included them with the maps. We will also publish the Assembly language routines of CALL LINK(PEEKV,... and CALL LINK(POKEV,... that you can load as an assembly file or with CALL LOAD's for use in Extended Basic if you have memory Expansion.

CONSOLE GROM >0000 - >17FF 8K Bytes

There are 3 GROM chips mounted in our consoles and each chip contains 6K bytes of programming or data. Each chip resides within an 8K address space so there is 2K of useless addresses per chip. GROM chips are a special type of ROM that is manufactured and patented by TI. GROM stands for Graphics Read Only Memory but TI also manufactures GRAM, or Graphics Random Access Memory, chips but they are not used in our consoles. The unique property of GROM is that it is auto incrementing ROM which means that every time it is accessed in automatically increments to the next address within itself. GROM mainly contains programs and routines written in GPL code. GPL is TI's own Graphics Programming Language that they are not too quick to give out any info on. GROM also contains large amounts of data that is pertinent to the system monitor and operation of our consoles. The 3 GROM chips in the console contain the following:

GROM 0 >0000 ->17FF The Title screen power up routine, Title screen character set, Standard character set, Lower case character set, Hi-Level cassette DSR messages and the Trig functions such as ATN, SIN etc.

GROM 1 >2000 ->37FF Vector tables for the Basic language, The ERROR statements for Basic and part of the Basic Interpreter.

GROM 2 >4000 ->57FF Part of the Basic Interpreter, the reserved word list and their associated token values.

GROM chips 3 through 6, 24K Bytes, reside in the Extended Basic cartridge and they contain the following.

GROM 3 >6000 ->77FF X-Basic vector tables, the ERROR statements for X-Basic and part of the X-Basic Interpreter.

GROM 4 >8000 ->97FF Part of the X-Basic Interpreter.

GROM 5 >A000 ->B7FF Part of the X-Basic Interpreter.

GROM 6 >C000 ->D7FF Part of the X-Basic Interpreter, the reserved word list and their associated token values.

Now lets add it all up like the other computer companies do!!!

CONSOLE MEMORY

```
 8K Console   ROM @>0000->1FFF          8K Console RAM @>8000->9FFF
18K Console  GROM @>0000->5800 *       16K VDP     RAM @>0000->3FFF *
-------------------------------        -------------------------------
26K of ROM/GROM Total                  24K of RAM total
```

CONSOLE MEMORY plus EXTENDED BASIC

```
 8K Console   ROM @>0000->1FFF
12K X-Basic   ROM @>6000->7FFF
18K Console  GROM @>0000->57FF *        8K Console RAM @>8000->9FFF
24K X-Basic  GROM @>6000->D7FF *       16K VDP     RAM @>0000->3FFF *
-------------------------------        -------------------------------
62K of ROM/GROM Total                  24K of RAM Total
```

Now add
MEMORY EXPANSION and the SPEECH SYNTHESIZER

```
 8K Console   ROM @>0000->1FFF
12K X-Basic   ROM @>6000->7FFF          8K Expan   RAM @>2000->3FFF
18K Console  GROM @>0000->57FF *        8K Console RAM @>8000->9FFF
24K X-Basic  GROM @>6000->D7FF *       24K Expan   RAM @>A000->FFFF
32K Speech    ROM @>0000->7FFF *       16K VDP     RAM @>0000->3FFF *
-------------------------------        -------------------------------
94K of ROM/GROM Total                  56K of RAM Total
```

and finally add
THE RS232 CARD and THE DISK CONTROLLER CARD

```
 8K Console   ROM @>0000->1FFF
 2K RS232 DSR ROM @>4000->4800 (aprox)
 8K DISK  DSR ROM @>4000->5FFF
12K X-Basic   ROM @>6000->7FFF          8K Expan   RAM @>2000->3FFF
18K Console  GROM @>0000->57FF *        8K Console RAM @>8000->9FFF
24K X-Basic  GROM @>6000->D7FF *       24K Expan   RAM @>A000->FFFF
32K Speech    ROM @>0000->7FFF *       16K VDP     RAM @>0000->3FFF *
-------------------------------        -------------------------------
104K of ROM/GROM Total                 56K of RAM Total
```

Boy do we have a lot of memory mapping to do!!

* Note: These items are memory mapped devices. You cannot PEEK or LOAD here from Extended Basic.

Here are a couple of programs that use CALL LOAD to change your screen display so you will need expansion memory to run them.

The first program places your computer in the Multi-Color mode and then generates patterns on the screen. Both this program and the one following it are fighting the Extended Basic system monitor in that the computer wants to return to a normal 32 column display in graphics 1 mode for Extended Basic. We presented them here for you to play with, they aren't what we would call truly useful but they are fun to look at.

After you have tried the first program replace lines 30 and 40 with some PRINT, DISPLAY AT and HCHAR or VCHAR statements of your own, you won't be able to see any text but it produces some interesting patterns. We will discuss Multi-Color mode more in future issues.

What this program does is to place the value needed for getting into Multi-Color mode into an address in the scratch pad ram and this address is a duplicate of VDP register 1. The operating system in our computer automatically copies whatever is in this address into VDP register 1 when a key is pressed that is why we have the CALL KEY statement in line 20. Once you press a key the computer will switch itself into Multi Color mode. If you stop the program with FCTN 4 (CLEAR) the System monitor will return the screen to its normal appearance.

You can use sprites in Multi-Color mode but the magnification is also in VDP register 1 so for now just use standard size sprites. When we map out VDP RAM we will included all the info on the different VDP registers.

```
10 CALL CLEAR :: CALL INIT :
: CALL LOAD(-31788,232):: PR
INT "PRESS ENTER"

20 CALL KEY(0,K,S):: IF S=0
THEN 20 ELSE CALL SCREEN(2)

30 CALL CLEAR :: FOR T=1 TO
7 :: FOR I=34 TO 126 :: PRIN
T CHR$(I);:: NEXT I :: NEXT
T

40 FOR I=34 TO 126 :: CALL V
CHAR(1,1,I,768):: NEXT I ::
GOTO 30
```

The following program also uses CALL LOAD to place you into Text Mode (40 Column). This program is also fighting the Extended Basic system monitor which thinks it is still in a 32 column display mode with 28 column PRINT lines.

We can get into text mode by using a similar method as the last program however we could not see any text. The foreground and background colors in Text mode are stored in VDP register 7 which is not directly accessible from Extended Basic. So what we did was to write a small assembly language program that switches the mode and sets the foreground and background color. Then we loaded this program and used CALL PEEK to take it apart so we could present it as CALL LOAD's for everyone that does not have the Editor Assembler command module.

Because of the AORG statement in the Assembly version we knew the the program would be in memory at address hex 3000 which is decimal 12288. We also had to load the LINK name for this program in the REF/DEF table which resides at 3FF8 since there is only one name in it. The rule for the REF/DEF table is that the name must be 6 characters in length, including trailing spaces, and have 2 bytes following the name that give the address in memory where the program starts. Don't forget that the first byte is multiplied by 256 and the second byte is added to it to form the word that indicates the address. In our case it is 48*256 = 12288 plus 0 = 12288 or hex >3000.

There is one more thing that MUST BE done in order to let the computer know that some thing is in the REF/DEF table. You must load the starting address of the REF/DEF table into hex 2004 or decimal 8196. Our REF/DEF table starts at 16376, 8 bytes up from the bottom of the table, 6 for the name and 2 for the address. So 16376 divided by 256 = 63.96875. 63 is the first byte so we will subtract 63 from 63.96875 and multiply .96875 times 256 to get 248 which is the second byte and that is how the first CALL LOAD on line 10 in the program came to be.

The second CALL LOAD at 16376 loads the LINK name and the starting address of our program. The CALL LOAD's on lines 20 and 30 are the Assembly language program. In the CALL LOAD on line 30 you will find the

number 244 at the beginning of the second line, this is the foreground and background color of the text on the screen you can play around with this value to change the colors without hurting the program. Once again we will discuss this value when we map out VDP RAM.

When you run this program things won't line up right on the screen because we are fighting that system monitor but some things of interest come into view on the bottom of the screen. Because Text Mode allows 960 character positions on the screen we are able to see part of VDP RAM that is just below the normal screen image table. The part that has been brought into view is called the Sprite Attribute Table, the Sound Table and the Crunch buffer. I hate to keep doing this but once again we will discuss these in depth with the VDP maps, for now just play with this and watch what happens.

While you are running the program it asks you to input a string and as you do the screen scrolls up but everything is offset because the system thinks it is in 32 column mode. After you play with this change line 50 and put all 28 sprites into motion by using a FOR NEXT loop and add line 60 to read GOTO 60 to keep the program running. You will not be able to actually see the sprites since they are not allowed in Text Mode. You will be able to see the bytes being updated in the Sprite Attribute table and this is the same thing that happens when you are in the normal mode with visible sprites.

For now just play around with it and we will keep looking for a way to defeat the system monitor in Extended Basic so we can have a true 40 column display for our programs in Extended Basic.

```
10 CALL CLEAR :: CALL INIT :
: CALL LOAD(8196,63,248):: C
ALL LOAD(16376,84,32,32,32,3
2,32,48,0)

20 CALL LOAD(12288,2,224,131
,224,2,1,240,129,216,1,131,2
12,216,1,140,2,6,193,216,1)

30 CALL LOAD(12308,140,2,2,1
,244,135,216,1,140,2,6,193,2
16,1,140,2,6,155)

40 CALL LINK("T")

50 INPUT A$ :: IF A$="C" THE
N CALL CLEAR :: GOTO 50 ELSE
 50
```

Here is the Assembly language version of the previous program's CALL LOADS on lines 10, 20 and 30.

```
Line Hex  Opcd Label     Opcode
Num. Add. Valu Field     Field

0001                     DEF T
0002 3000                AORG >3000
0003 3000 02E0  T        LWPI >83E0
     3002 83E0
0004 3004 0201           LI R1,>F081
     3006 F081
0005 3008 D801           MOVB R1,@>83D4
     300A 83D4
0006 300C D801           MOVB R1,@>8C02
     300E 8C02
0007 3010 06C1           SWPB R1
0008 3012 D801           MOVB R1,@>8C02
     3014 8C02
0009 3016 0201           LI R1,>F487
     3018 F487
0010 301A D801           MOVB R1,@>8C02
     301C 8C02
0011 301E 06C1           SWPB R1
0012 3020 D801           MOVB R1,@>8C02
     3022 8C02
0013 3024 069B           BL *R11
0014                     END
```

If you use the number conversion program from Vol 2 of our newsletter on the hex numbers in the Opcd Valu (Opcode Value) column and convert them into decimal you will find that they are the same as the ones used in the CALL LOAD statements. Play around with this and become familiar with it because we plan on presenting more Assembly language programs that will be loaded with CALL LOAD along with the Assembly version so everyone with expansion memory can benefit from them.

DRAW PROGRAM

The following program does not require expansion memory and it is still a lot of fun to play with. It does however require Extended Basic and a joystick to use it. This program allows you to draw on the screen using only the joystick and the fire button.

When you run the program the screen will go black and fourteen different colored boxes will be displayed at the top of the screen. In between each colored box is a black box and it can be used as a color or to erase a small part of your drawing. To pick up a color move the white dot with the joystick on top of the color you want on the top row and press the fire button. The color you have picked up will be display in the upper left hand corner of the screen.

Now you can move the white dot to any location on the screen and wherever you want to put a colored block just press the fire button, you can also hold down the fire button and draw as you move the white dot around. The program will also allow you to wrap around the screen in all directions so you can easily go from the bottom of the screen to the top to get a new color.

When you are ready for a new color just go back up to the top of the screen and pick it up. If you want to erase your drawing from the screen just move the white dot up to the top of the screen, any where on the first row, and press the 1 key. After the screen is erased your white dot will be at its starting position but it will still be loaded with the last color you were using.

There are a lot of items you could add to this program since there is so much memory left. You could add IF K=7 THEN 180 ELSE .... to the beginning of line 170. Then when your white dot is on the top row of the screen and you press the 2 key, keycode 7 on the split keyboard scan, you will leave this program and jump down to your own routine on line 180. How about a routine that saves the screens out to disk or cassette using CALL GCHAR in FOR NEXT loops. Maybe you could jump down to execute a screen dump utility, if you have one, or whatever else you can come up with.

```
100 CALL CLEAR :: CALL SCREE
N(2):: K=2 :: W=32 :: FOR S=
88 TO 136 STEP 8 :: CALL CHA
R(S,"FFFFFFFFFFFFFFFF00"):: 
NEXT S

110 CALL CHAR(42,"00003C3C3C
3C"):: CALL COLOR(1,14,2,8,1
3,3,9,4,5,10,6,8,11,7,9,12,1
0,11,13,12,14,14,15,16)

120 FOR S=88 TO 136 STEP 8 :
: DISPLAY AT(1,K):CHR$(S)&"
"&CHR$(S+1):: K=K+4 :: NEXT
S

130 CALL HCHAR(2,1,32,736):: 
 CALL SPRITE(#1,42,16,17,121
):: Y=3 :: X=16 :: CALL SOUN
D(-100,660,9)

140 CALL JOYST(1,K,S):: X=X+
SGN(K):: Y=Y-SGN(S):: IF Y>2
4 THEN Y=1 ELSE IF Y<1 THEN
Y=24

150 IF X>32 THEN X=1 ELSE IF
 X<1 THEN X=32

160 CALL LOCATE(#1,Y*8-7,X*8
-7):: CALL KEY(1,K,S):: IF S
=0 THEN 140 ELSE IF Y>1 THEN
 CALL SOUND(-90,-2,15):: CAL
L HCHAR(Y,X,W):: GOTO 140

170 IF K=19 THEN 130 ELSE CA
LL GCHAR(Y,X,W):: CALL SOUND
(-90,880,7):: CALL HCHAR(1,2
,W):: GOTO 140
```

This DRAW program was programmed using some of the routines and formulas from the Smart Programming Guide for Sprites. We wanted to present it here in hopes that it might help you to better understand how to group together and modify some of the different routines from the book.

You will find the documentation for parts of this program in the Formula section and in JOYST 3. By using KEY 3 you can add keyboard inputs to it. I hope you enjoy using and modifying it.

5TH 1- =FORTH

AN INTRODUCTION TO TI FORTH

Forth is a very fast and powerful language that Texas Instruments will be releasing to the TI 99/4A Users' Groups sometime in Feb. If you don't belong to a local users' group and you would like to bring out ALL the power of your 99/4A I would strongly recommend that you join one. For the cost of joining a group and the costs of duplicating the Forth manual and diskette you will be able to write and run some very powerful programs. To program in TI Forth you will need the following equipment. A console, monitor or TV, the Editor Assembler command module, 32K Memory expansion, at least one disk drive and of course a copy of TI FORTH.

Forth opens up the entire computer to your programming imagination and it is much easier to program in than Assembly language. Anything you can do in Assembly language you can do in Forth from writing a simple program to writing rapid action, interrupt driven, bit mapped games or your own programming language.

It is a word orientated language similar to LOGO in that you define a word as a procedure and then you can use that word to help define the next word and so on. That is how you build a program in Forth. The words that you define are added to Forth's resident vocabulary, which in the case of TI Forth is pretty extensive already.

TI Forth supports ALL of the VDP modes and it even adds two more of its own. Besides Graphics mode (32 Column), Text mode (40 Column), Multi-Color mode and Bit-Mapped mode TI added SPLIT and SPLIT2. SPLIT mode is a Bit-Mapped screen with 8 lines of text at the bottom of the screen. Perfect for Graphic adventure games and plotting routines. SPLIT2 mode is also a Bit-Mapped screen but it allows 4 lines of text at the top of the screen, this one is great for programs that need Bit-Mapped graphics and only a little text. All of these modes can be entered into by typing in a one word command such as TEXT, SPLIT or GRAPHICS. Bit-Mapped screens support words such as LINE, DOT, DRAW and UNDRAW. With SPLIT and SPLIT2 modes you can type in commands and watch them work.

TI Forth also supports 32 Sprites in all of the VDP modes except TEXT mode. Many of the Sprite commands are very similar to the ones you are currently using in Extended Basic except they operate MUCH faster. At last a COINC that works right. Some of the commands for Sprites are; SPRITE, SPRCOL, COINC, COINCALL, COINCXY, DELSPR, DELALL, SPCHAR, SPRPAT, SPRPUT, SPRGET, MOTION, SPRDIST and SPRDISTXY. Forth runs so fast that you could easily write a few of your on words to add to this list, say for checking COINC against a graphic character or for finding out which two sprites have COINC. How about calling them COINCHAR and COINCWHCH.

Unlike Assembly Language you can type in commands or define words in the immediate mode and see how they operate before you include them in your program. Forth is a complied language so when you are defining a word in the immediate mode and you press ENTER it is instantly compiled and placed in the dictionary. Then you just type in the new word and watch it work.

When you are ready to start writing your program you can enter the edit mode by telling it the screen number on the disk and EDIT, such as 30 EDIT <ENTER>. TI Forth will then bring up a 64 column, standard for Forth, windowed screen for you to type in your programs on. After you have entered your program just type in FLUSH and Forth will write it out to the disk at that screen. Forth allows 90, 1K Byte screens per single sided, single density diskette. When you what to load and run your program just type in 30 LOAD and it will load and compile your program into the dictionary. It will also auto-run it if you set it up that way or you can type in a one word command to run it. TI Forth also has a 64 column editor that is not windowed and uses SPLIT mode, Bit-Mapped, to give you 64 columns on the screen at the same time.

If you haven't noticed by now I am really excited about Forth's power, speed and ease of programming, this is what a computer is supposed to be like. If you've tried programming with Assembly and haven't gotten too far look into Forth, you'll love it. Even if you are an avid Assembly programmer you'll like Forth since you can write assembly code right in line with the rest of your program and Forth will assemble it.

WHATS NEW FROM CES

There was a lot of good news for TI 99/4A owners at CES. Before I get started let me explain what CES is. It stands for Consumer Electronics Show and it is held twice a year. Once in the summer around June at Chicago and also in January at Las Vegas. The show was very large and encompassed the entire Las Vegas convention center as well as parts of three major hotels. We walked around for three days and still didn't see it all. There wasn't as much on computers as there was on stereo's and telephones but there were a lot of software companies in the computer section.

Most of the software companies that we talked with said they planned on continued support for the 99/4A. A few of them talked about new releases that will be coming out over the next 9 to 10 months. Atari soft has has the following releases planned, some of which are already out. Pac-Man, Donkey Kong, Centipede, Dig Dug, Defender, Robotron: 2084, Pole Position, Ms Pac-Man, Jungle Hunt, Moon Patrol, Joust, Shamus, Protector II and Picnic Paranoia. Tigervision plans on releasing six more titles to compliment its current Miner 2049er release. The titles are Springer, Scraper Caper, Espial, Changes, Sky Lancer and Super Crush.

In talking with Scott Adams he said that they plan on continued support for the 99/4A. At the time we talked with him he stated that they had further negotiations with TI before he could say anything definite. There were also quite a number of other companies such as Scott Foresman, DLM, Broderbund and Sieara On Line, to name a few, who had further negotiations with TI planned. All of the companies we talked with except CBS games and Milton Bradley said they planned on continued support for the TI 99/4A !!! That was nice to hear.

We also looked at Oscar, the bar code reader from DataBar, for the 99/4A while we were there. They didn't have a working version of any of the software packages that will be released in bar code form for use with their reader, but for around 80 dollars with some software included it looked pretty interesting. It should be available very soon at your local dealer.

While we were at the show we stopped in at CorComp's hospitality suite to look at their P-Box prototype. We also heard a rumor that they will bring out a baby P-Box prior to the full P-Box. The baby P-Box will probably hold up to 4 cards. Your disk drives will have to be external and have their own power supply and case. We haven't heard any prices at this time.

We also spotted a very nice letter quality daisy wheel printer made by Dynax. The Dynax model DX 15 has a 15 inch carriage that is friction feed and prints at 13 characters per second ,13 CPS. It also comes standard with a parallel interface and a 3K Byte buffer. The option list for this printer includes a tractor feeder, cut sheet feeder and an optional keyboard that turns it into an electric typewriter!!

It looks like Ron Wells will be carrying this printer for around 499.95 plus shipping, he seems to carry everything. If you would like further information on the printer you can contact him or Diane at (714) 983-2878. His address is 5523 San Jose, Montclair, CA 91763. Oh yes, they also have a special on Signalman Mark III modems right now. This modem is made for the 99/4A and it comes with a free one year subscription and sign on to the Source, which is valued at $100.00. They are selling the modem with the subscription for 84.95 plus shipping.

While we are talking about modems we would like to ask everyone that has ANY information about ANY Bulletin Board Systems for TI computers to send us the info. We are planning on running a complete listing of BBS for TI users in a future issue. Also when you write please included as much info as possible about the BBS, such as sign on fee, types of data , phone number and public domain access codes etc. With your help we should be able to compile a nice list from around the world.

Our TI PC column will start next month, we ran a little short on space this month.

In closing for this month we would like to thank everyone who subscribed and all of the TI 99/4A Users' Group's for your continued support, it wouldn't have been possible without YOU!

S U B S C R I P T I O N I N F O R M A T I O N

THE SMART PROGRAMMER - a monthly 16+ page newsletter published by **MILLERS GRAPHICS**
U.S. 12.50 year - Foreign Surface Mail 16.00 year - Foreign Air Mail 26.00 year

To subscribe send a Check, Money Order or Cashiers Check, payable in U.S. currency

TO: **MILLERS GRAPHICS**
1475 W. Cypress Ave.
San Dimas, CA 91773

MILLERS GRAPHICS
1475 W. Cypress Ave.
San Dimas, CA 91773

3R